A Framework for Computation-Memory Algorithmic Optimization for Signal Processing

Gene Cheung, *Member, IEEE,* and Steven McCanne

***Abstract*—The heterogeneity of today's computing environment means computation-intensive signal processing algorithms must be optimized for performance in a machine dependent fashion. In this paper, we present a dynamic memory model and associated optimization framework that finds a machine-dependent, near-optimal implementation of an algorithm by exploiting the computation-memory tradeoff. By optimal, we mean an implementation that has the fastest running time given the specification of the machine memory hierarchy. We discuss two instantiations of the framework: fast IP address lookup, and fast nonuniform scalar quantizer and unstructured vector quantizer encoding. Experiments show that both instantiations outperform techniques that ignore this computation-memory tradeoff.**

***Index Terms*—Computation theory, memory management, packet switching, signal processing, vector quantization.**

I. Introduction

IF THE computer evolution has matured to a stage where computers are ubiquitous and homogeneous, and improvements are asymptotic, then implementation of an algorithm needs only be painstakingly hand-coded once for optimal performance. Unfortunately, computers continue to progress at an exponential rate, and computing environments are extremely diverse. Clearly, hand-coding an algorithm for every possible platform is impractical. A fundamental question surfaces: how to re-target an algorithm onto different machine platforms optimally *and* automatically? In light of this problem, formal techniques on flexible software synthesis and code generation [1]–[4] have been extensively studied. [1], [2] are examples of retargetable compilers that can generate efficient code given a description of the machine architecture. [3] proposes a graph-based optimization technique to generate digital signal processor (DSP) address code. [4] mixes models of computation—imperative programming language such as C and signal processing specific model such as synchronous data flow (SDF), for optimal DSP compilation while offering intuitive user appeal.

What is common among the above *compiler-centric* approach, is that because the starting point of the optimization is still a fairly generic language, machine-dependent high level algorithmic optimization is often hidden from the optimizer and left unexploited. Contrasting with this approach is the *signal processing-centric* approach [5], [6], which optimally trades off distortion of compressed signals with computation of a particular compression algorithm by intelligently "tweaking" the algorithm. Such optimal tradeoffs are the first formal analyzes of signal processing algorithm tuning for machines with different computational budgets.

Similar to other signal processing-centric studies, our work [10]–[12] focuses on algorithmic level optimization—in our case, minimizing running time of algorithm—in a machine dependent fashion. Minimizing running time is important when, for example, one wants to determine if a particular machine is capable of satisfying a set of real-time constraints required by a particular signal processing algorithm. However, unlike previous works that minimize running time by minimizing the number of computational units for a given performance threshold, we search for the optimal computation-memory tradeoff to minimize running time. More precisely, we set up and solve the following optimization problem. Given an algorithm, we first define a search space of *configurations*, where each configuration is a particular implementation of the algorithm. Each configuration uses a different mixture of computation and memory. At one extreme, a configuration can use only memory and no computation—using a lookup table, a given input can be directly mapped to the corresponding *precomputed* output. This is possible because each input value is represented by a fixed number of bits in a digital computer, and so the set of possible input to an algorithm is countably finite. However, the set of input values is nevertheless very large, and so given a machine has small and finite memory, this direct-map configuration is usually not practical.

At the other extreme, a configuration can use only computation and no memory—for every input, the corresponding output is *computed* on-the-fly with no lookup of pre-computed values. Storing nothing in memory means no computing effort can be amortized across inputs, and every output must be computed from scratch. Given that the objective is to optimize execution speed of the algorithm, this configuration underutilizes memory of a machine and often is suboptimal.

Intuitively, the optimal configuration of an algorithm for a given machine lies between these two extremes: one that divides the processing so that the optimal subset is implemented as memory retrievals of precomputed values (precompute), and the other is implemented as on-the-fly computations (compute). For each algorithm, deciding which is the optimal configuration depends on two pieces of information: 1) the particulars of the algorithm, which reveals ways in which the algorithm

Manuscript received July 28, 2000; revised March 19, 2002. This work was completed while the authors were with the Department of Electrical Engineering and Computer Sience, University of California, Berkeley. The associate editor coordinating the review of this paper and approving it for publication was Dr. Francky Cathoor.

G. Cheung is with HP Laboratories Japan, Tokyo 168-0072 Japan (e-mail: gene-cs.cheung@hp.com).

S. McCanne is with the Department of Electrical Engineering and Computer Sience, University of California, Berkeley, CA 94720 USA.

Digital Object Identifier 10.1109/TMM.2003.811625

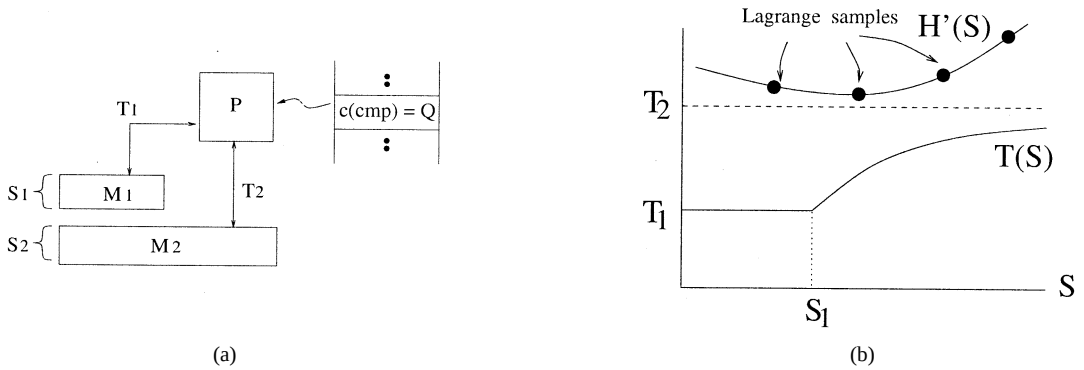


Fig. 1. Proposed machine model and functions of optimization framework. (a) Dynamic memory model. (b) Memory access function T(S) and Lagrange sampling function H'(S).

can be implemented, and hence implicitly defines the search space of configurations; and, 2) the memory structure of the to-be-implemented machine, which defines the feasibility and cost of each configuration. The major contribution of this paper is to formalize and solve this optimization based on this information. In particular, we develop an optimization framework that when instantiated, finds a near-optimal implementation for a given algorithm. Each instantiation of the framework for a particular algorithm is called a *program*. The algorithms we focus on in this paper are 1) **IP Address Lookup**—In the current Internet architecture, packets of information called IP packets are forwarded from source to destination via a network of routers. At each router, a packet's IP destination address is matched against the prefixes of a routing table—the packet is subsequently forwarded to the outgoing link corresponding to the longest matched prefix. Toward the goal of optimal network performance, designing a fast address lookup implementation is an important problem. 2) **Scalar and Vector Quantizer Encoding**—Quantization is a lossy signal compression technique where a many-bit representation of a signal is mapped to a few-bit representation. Quantizer encoding—the many-to-few bit mapping—can be a time-consuming process, especially for unstructured vector quantizer of high dimensions. Designing a fast encoder implementation is a popular research problem.

The outline of the paper is as follows. In Sections II and III, we review the machine model and its associated optimization framework in [11] respectively. In Section IV, we discuss how an instantiation of the framework finds a near-optimal implementation for IP address lookup. In Section V, we discuss how a different instantiation of the framework finds an efficient implementation of scalar and vector quantizer encoders. We present results in Section VI. Finally, we conclude and discuss future work in Section VII.

II. Dynamic Memory Model (DMM)

Modern processors use hierarchical memories [7] to enhance performance, where small, fast memories are located near the CPU and larger, slower memories are situated further away. Memory design remains an active area of research [8], [9] as it continues to have a significant impact on performance. While newer proposals vary in details, the basic philosophy of having a memory hierarchy remains the same, and we will focus on this feature in developing our model.[1] Given a hierarchy of memory, the execution speed of a machine instruction that accesses memory depends on the level of memory referenced. The machine model we adopt in Fig. 1(a), called Dynamic Memory Model (DMM), reflects this characteristic. If the processor $\mathbf{P}$ looks and finds a datum residing in level 1 memory $\mathbf{M}_1$, it incurs a *hit time* T_1—this includes the time needed to determine if it is a hit/miss and the time for datum retrieval. If $\mathbf{P}$ looks and cannot find the desired datum in $\mathbf{M}_1$, then it pays a *miss penalty* T_2—this includes the time for loading this datum from $\mathbf{M}_2$ to $\mathbf{M}_1$ and the time for delivering datum from $\mathbf{M}_1$ to $\mathbf{P}$. If the instruction does not involve memory access, then the execution time depends on the complexity of the instruction itself. For example, we denote the cost of a logical comparison (`cmp`) as Q. We assume the sizes of $\mathbf{M}_1$ and $\mathbf{M}_2$ are S_1 and ∞ respectively.

To deduce deterministically the datum retrieval time at time t during the execution of an algorithm, one will require knowledge of what data are residing where in the hierarchical memories at t. This will require knowledge of both the processor's caching policy (direct mapped, set associative or fully associative [7]) and the data access history from time zero to t. In a general signal processing setting where the input data is not known until the algorithm is executed, choosing an optimal implementation a priori deterministically by tracking all this information is simply not possible. Instead, we take a probabilistic approach and approximate each memory access cost as follows.

Suppose the size, S, of data structures used by an implementation of an algorithm (configuration) is $\leq S_1$. Then the access time of a desired datum, $T(S)$, is T_1, since all data structures can be loaded into $\mathbf{M}_1$. If $S > S_1$, then we do not know whether the desired datum resides in $\mathbf{M}_1$ or $\mathbf{M}_2$. In this case, we estimate the access time as follows. At any given time, a fraction $((S_1)/(S))((S - S_1)/(S))$ of the total number of data blocks will be in $\mathbf{M}_1$ $(\mathbf{M}_2)$. So assuming all pieces of data are equally

[1]Other details of memory architecture are also important to algorithm performance. However, here we focus on extracting common features across different memory architecture in constructing a mathematical model, so that implementations of algorithms can be optimized for different machines. Looking solely at the hierarchical structure seems like a logical first step. We will discuss the design of more accurate mathematical memory models in the last section.

probable, we can estimate memory access cost $T(S)$ of a data memory retrieval as

$$TS(S) = \begin{cases} T_1 & \text{if } S \le S_1 \\ \left(\frac{S_1}{S}\right) T_1 + \left(\frac{S-S_1}{S}\right) T_2 & \text{otherwise.} \end{cases} \quad (1)$$

See the bottom curve of Fig.1 (b) for an illustration of $T(S)$. In some applications, the input stochastic model to the algorithm is input-to-input independent; in such case, each memory access can only be estimated using (1). In other applications, like IP address lookup discussed in Section IV, the input model is Markovian; in such case, each repeated memory access to the same datum yield access cost T_1. More details on the relationship between input models and the memory model are discussed in Sections IV and V.

III. Optimization Framework (DMMOPT)

Using DMM, we can formally define the optimization problem of finding an optimal configuration for an algorithm as follows. Let $\mathcal{L}$ denotes the set of configurations in the search space. Each configuration $l \in \mathcal{L}$ uses a set of data structure in its implementation. Given the set of data structure, we assume the data memory size $R(l)$ of a configuration l can be easily determined. See Section IV and V for details of how $R(l)$ is determined given l for practical applications.[2]

We evaluate the execution cost of a configuration l as follows. First, the data memory size $R(l)$ translates to a memory access cost $T(R(l))$ using (1). Knowing the access cost, we can evaluate the execution cost of l, $H_{T(R(l))}(l)$. The optimization problem is

$$\min_{l \in \mathcal{L}} \left\{ H_{T(R(l))}(l) \right\}. \quad (2)$$

Solving (2) is difficult in general (for the VLC decoding problem, see [11] for a formal proof of NP-hardness). The reason is twofold. 1) while the cost of a memory access is not known till the entire configuration is constructed, the optimal construction of a configuration depends on the cost of memory access—a chicken-and-egg problem, and 2) because the cost evaluation depends on nonlinear function $T(S)$, the problem is nonlinear. Instead of solving (2) directly, we dissect it into easier pieces.

A. Problem Transformation

Suppose we know a priori that the total data structure size of the optimal configuration l^* is S^*. To find l^*, we only need to search the subset of $\mathcal{L}$ with total size S^*. (2) is then the same as:

$$\min_{l \in \mathcal{L}} \left\{ H_{T(R(l))}(l) \right\} = H'(S^*) \quad (3)$$

where we define $H'(S)$ as:

$$H'(S) = \begin{cases} \min_{l \in \mathcal{L}} \left\{ H_{T(S)}(l) \right\} \\ \text{s.t. } R(l) = S \\ \text{undefined} \end{cases} \quad \begin{array}{l} \text{if } \{l \in \mathcal{L} | R(l) = S\} \neq \emptyset \\ \text{o.w.} \end{array} \quad (4)$$

and $H_{T(S)}(l)$ is the cost of l when the memory access cost is fixed at $T(S)$. We call $H'(S)$ the *sampling function*. We additionally denote the set of S values where $H'(S)$ is well defined in (4) as $\mathcal{V}$. Solving $H'(S^*)$ seems easier, since we have eliminated the problems of mutual dependency and nonlinearity. However, we do not know S^* a priori, and so we need to search through all definable values of S in $H'(S)$ for S^*:

$$\min_{l \in \mathcal{L}} \left\{ H_{T(R(l))}(l) \right\} = \min_{\forall S \in \mathcal{V}} \left\{ H'(S) \right\}. \quad (5)$$

Graphically, for each definable S, we solve (4) and obtain a sample point on $H'(S)$. S^* is the minimum point on sampling function $H'(S)$. See the top curve of Fig.1(b) for an illustration of $H'(S)$. We are now faced with three new difficulties: 1) it is unclear how to determine which S value is definable—$S \in \mathcal{V}$, 2) solving (4) for all $S \in \mathcal{V}$ can be expensive, and 3) solving (4) given S itself is still hard since it is a constrained problem.

B. Lagrangian Approach for DMM

Focusing only on the third difficulty—solving (4) given S, we take the conventional approach of solving its corresponding Lagrangian instead:

$$\min_{l \in \mathcal{L}} \left\{ H_{T(S)}(l) + \lambda R(l) \right\}. \quad (6)$$

The two problems, (4) and (6), are related: it can be shown [13] that if there exists a Lagrange multiplier λ, such that the optimal solution to (6), l^o, satisfies $R(l^o) = S$, then l^o is also optimal to (4). Note that if this is the case, then $S \in \mathcal{V}$ by definition.

In general, solving the corresponding unconstrained problem (6) is easier than the original constrained one (4). The problem is that for a given value S, there may not exist a multiplier λ such that the optimal solution to the Lagrangian (6), l^o, has the property $R(l^o) = S$. In this case, we cannot even be sure if S is definable.

In such a case, we propose the following procedure called the *iterative projection method* that converges to a definable value S *and* a multiplier λ such that $R(l^o) = S$:

Iterative Projection Method:

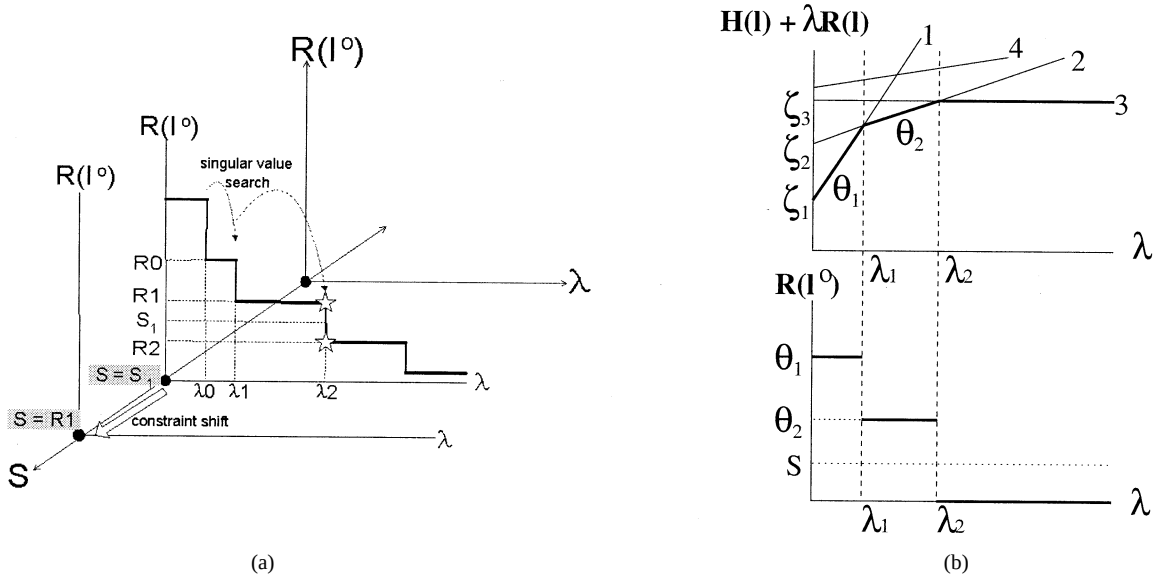
1) Initialize S.
2) Iteratively solve (6), adjusting λ each time, such that $R(l^o)$ is as close to S as possible while keeping $R(l^o) \ge S$.
3) If $R(l^o) = S$, done. Else, let $S := R(l^o)$ (*constraint shift*), goto step 2.

See Fig. 2(a) for an illustration. Because $R(l^o)$ is inversely proportional to λ, a simple strategy of adjusting λ in step 2 is to use binary search on the real line. Alternatively, a more efficient strategy, called singular value search, can be employed. We will discuss this strategy next.

C. Singular Value Search

Recall in step 2 of the Iterative Projection Method, we need to adjust the multiplier value and solve (6) iteratively, until $R(l^o) - S$ is minimized while $R(l^o) \ge S$. Consider the example in Fig. 2(b). The Lagrangian cost of every configuration l in search space $\mathcal{L}$, $H(l) + \lambda R(l)$, is represented as a linear function of multiplier λ in the top graph. The bottom graph plots the slope of the optimal Lagrangian cost function given λ, or simply $R(l^o)$. Notice as λ increases, the optimal configuration changes from l_1 to l_2 to l_3, and the corresponding $R(l^o)$ changes from θ_1 to

[2]Small, trivial examples available in these sections may facilitate understanding better than just formal definitions.



(a) (b)

Fig. 2. Lagrangian approach. (a) Singular value search and iterative projection method. (b) Lagrangian cost and size versus multiplier

θ_2 to 0. Also note that l_4 is never an optimal configuration for any multiplier value.

At special multiplier values, called *singular values* in [13] and labeled λ_1, λ_2 in Fig. 2(b), there are two (or more) configurations that are simultaneously optimal. For example, at λ_1, l_1 and l_2 are both optimal. Using similar argument as [13], one can show that by solving (6) *only* at singular values, we can discover all solutions to (6) for all multiplier values. Moreover, at the particular singular value where the slopes of the two optimal solutions span the constraint S, we can conclude that these two are the solutions with $R(.)$ closest to S from above and below, among all Lagrangian solutions. In the example, at λ_2, l_2 and l_3 are simultaneously optimal, and given they are the closest Lagrangian solutions from above and below, l_2 is the solution we are looking for.

Given we have an optimal solution l^o to (6) for a particular multiplier value λ_0, it turns out finding neighboring singular values to λ_0 is easy (to be shown in the two instantiations). Since $R(l^o(\lambda))$ is inversely proportional to λ, we can iteratively step through neighboring singular values in the direction of S until the ending condition—singular value where the two simultaneously optimal solutions have $R(.)$ spanning the constraint S—is met. This is called *singular value search*. In Fig. 2(b), we first initialize multiplier value to λ_0 and solve (6). We then step to singular value λ_1, then λ_2, upon which we have reached the ending condition.

D. Lagrangian Sampling

Instead of searching for all $S \in \mathcal{V}$ in (4), by finding solutions to (6) using the iterative projection method, we are actually only sampling a small number of points on $H'(S)$, since the method converges to a subset of points no matter what S is initialized to. We call this phenomenon *Lagrangian sampling*, since each sample point is a solution to the Lagrangian (6). By sampling, however, we may not be able to find the optimal solution $H'(S^*)$; we use the following theorem to bound the error of neighboring sample points from a local minimum point. The proof is sketched out in Appendix.

Theorem III.1 (Lagrangian Sampling Error Theorem): Let l^* be a locally optimal solution to (2) such that $S^* = R(l^*) \in [S^1, S^2]$, and $H'(S^1)$, $H'(S^2)$ are the two neighboring Lagrangian sample points on $H'(S)$. When S is initialized to S_1 in step 1 of iterative projection method, we can find an optimal solution to (6), l^B, such that one of the **Bounding Conditions** is satisfied: 1) $(R(l^B) \geq S^2)$ and $(\lambda \geq 0)$. 2) $(R(l^B) \leq S^1)$ and $(\lambda < 0)$. 3) $(S^1 \leq R(l^B) \leq S^2)$ and $(\lambda = 0)$. The cost of the locally optimal solution l^* is lower bounded by l^B, i.e.: $H'(S^*) = H_{T(S^*)}(l^*) \geq H_{T(S^1)}(l^B)$.

E. Optimization Framework for DMM—DMMOPT

Having developed the above concepts, the optimization framework associated with DMM called DMMOPT—one that guides us in constructing a *program* that finds a near-optimal implementation to (2) with a posteriori error bound—is straightforward. In a nutshell, we construct a program by instantiating the following procedure for each algorithm:

> Given parameters of the machine model (DMM) and the search space of configurations $\mathcal{L}$ of the algorithm, construct $H'(S)$ by obtaining Lagrangian sampling points. Each Lagrangian sampling point is obtained with the iterative projection method. Among the sampling points, we pick the smallest point as our operating point.

The global error bound is the difference between the best performance sample point and the best performance local bound of all pairs of neighboring sample points.

How the Lagrangian (6) is solved depends on the search space of configuration $\mathcal{L}$ of the algorithm being optimized. We will begin with the IP address lookup algorithm in the next section.

IV. IP Address Lookup (VLC Decoding)

The IP address lookup problem is the problem of efficiently finding the longest prefix in a routing table of a network router that matches the destination address of the IP packet. The entry

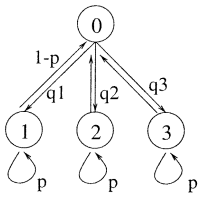


Fig. 3. Markov model for address prefixes.

corresponding to the longest matched prefix determines the output interface through which to forward the packet toward its ultimate destination.

The existing art in this problem domain is quite extensive [14]–[18]. However, none of the previous works exploit the memory hierarchy of the underlying processor as a formal optimization. We will do exactly that in this section by using the machine model DMM and optimization framework DMMOPT discussed in Section II and III respectively. We first discuss our proposed Markov model that models the correlation between prefixes of consecutive IP packets. We then discuss the transformation that transforms the longest prefix match problem to the variable-length code (VLC) decode problem. We then discuss solution to the VLC decode problem.[3]

A. Markov Model for Packet Prefixes

During a typical TCP connection, a burst of packets are sent back-to-back along the same route to the same destination. This results in a sequence of packets with the same destination IP address, meaning the same longest prefix requires consecutive lookups at the router. To model this dependency, we have constructed the simple Markov model shown in Fig. 3. Each prefix in a routing table is represented by a state—state 1, 2, 3 in the figure. In addition, there is an initial state 0. Starting at state 0, we enter state i with probability q_i. This represents a packet with longest prefix i has arrived at the router. With probability p, we return to the same state, representing the case when the next packet also has longest prefix i. With probability $1-p$, we return to initial state 0, and a new prefix is selected. The expected number of packets persisting in the same state[4] is $1/(1-p)$.

B. Transformation to VLC

Prefix set found at a routing table differs from variable-length code (VLC) in signal processing such as Huffman code in one major respect: prefixes in routing table are not prefix-free. In other words, a prefix entry can be a prefix of another prefix entry (see Fig. 4(a) for an example). If we represent the prefix set as a binary tree,[5] as shown in Fig. 4(b), then there may exist prefixes that are internal nodes and not leaves of the tree. Instead of operating on this tree to find the longest prefix, which requires backtracking, we perform *leaf pushing* [18] to convert it to *prefix-free* binary tree, shown in Fig. 4(c). In the example,

prefix	action
0	c
000	a
001	b
10	d
11	e

(a)

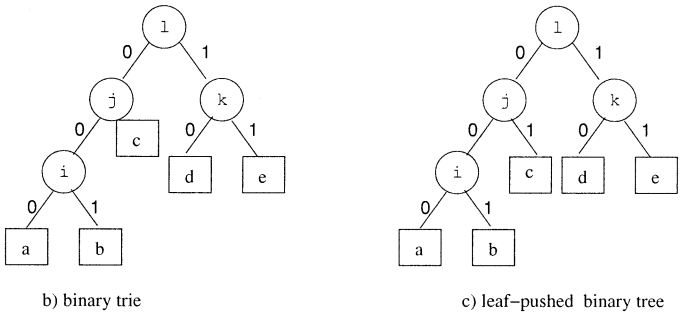


Fig. 4. Representation of a prefix set.

prefix 0 is first expanded to prefix 01, since address with prefix 01 means it has a longest prefix of 0 in the prefix set. We will assume such a conversion is first employed as a pre-processing stage for IP address lookup to convert prefixes to prefix-free VLCs. In the next section, we will discuss how the VLC decode problem is formally defined.

C. Problem Formulation

Given a set of prefix-free VLCs, the problem is then to find a configuration l that is fast for the particular processor's memory hierarchy, modeled by DMM. We choose the search space of configurations $\mathcal{L}$ to be configurations that use a mixture of two VLC decoding operators—lookup table and programmed logic. A n-bit lookup table—using up 2^n data memory—entails data retrieval in memory, hence the execution cost depends on the total size of data in memory. We assume programmed logic requires no data memory and has execution cost Q.

To impart intuition, we first consider two examples of VLC decoding configurations. To decode the set of prefixes in Fig. 4, two configurations are constructed in Fig. 5: 1) a programmed logic is followed by either a 2-bit table lookup or another programmed logic; 2) a two-bit table lookup is followed conditionally by a programmed logic. Graphically, we denote a programmed logic at a node with dark branch arrows and a lookup table by shading the node.

To determine the average decoding time of these configurations, we find the size of data structures in memory $R(.)$ for both configurations to be 4 (one 2-bit lookup table). We can now write the execution time of the first configuration l_1, denoted as $H_{T(4)}(l_1)$, as follows:

$$\begin{aligned} H_{T(4)}(l_1) &= (q_a+q_b+q_c)\left[Q+T(4)+\left(\frac{p}{1-p}\right)(Q+T_1)\right] \\ &\quad +(q_d+q_e)\left[2Q+\left(\frac{p}{1-p}\right)2Q\right] \\ &= (q_a+q_b+q_c)(Q'+T')+(q_d+q_e)(2Q') \end{aligned} \quad (7)$$

where $T'=T(4)+((p)/(1-p))T_1$, and $Q'=((1)/(1-p))Q$. We can rewrite the execution cost in terms of the probability

[3]The same optimization can be used to optimize VLC decoding, such as Huffman decoding, in signal processing.

[4]The expected number is $(1-p)+2p(1-p)+3p^2(1-p)+\ldots=((1-p)/(p))\sum_{i=1}^{i=\infty} ip^i = ((1)/(1-p))$.

[5]By representing the prefix set as a binary tree, we are implicitly restricting ourselves to decoding algorithms that decode sequentially from left to right. This is reasonable since all prefixes are left-aligned. Further, by allowing bits to be decoded in any order causes the optimization problem to be NP-hard [11].

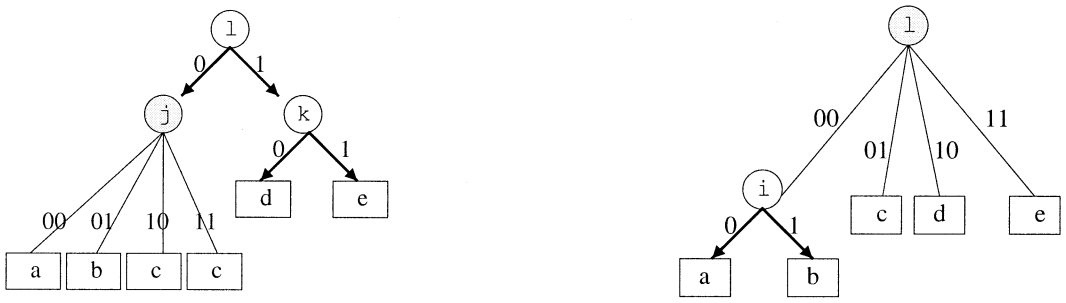


Fig. 5. Examples of prefix decoding using 1-bit logic and table lookup. (a) Prefix decoding configuration 1 and (b) prefix decoding configuration 2.

flow of the internal nodes. For example, probability flow of node j is $w_j = q_a + q_b + q_c$. We can now write $H_{T(4)}(l_1)$ as:

$$H_{T(4)}(l_1) = w_l Q' + w_j T' + w_k Q' \tag{8}$$

Similarly, the execution cost of the second configuration, l_2 will be $H_{T(4)}(l_2) = w_l T' + w_i Q'$.

We are now ready to formalize the optimal VLC decoding problem, called VLC-OPT, as follows.

Optimal VLC Decode Problem: VLC-OPT

Given: i) Parameters of DMM: S_1, T_1, T_2; ii) A set of VLCs and their associated probabilities. What is an optimal configuration so that the average decoding time is minimized? Mathematically, we write: $\min_{l\in\mathcal{L}}\{H_{T(R(l))}(l)\}$, where $\mathcal{L}$ is the set of configurations using lookup table and programmed logic that decodes the given prefix set correctly.

A proof in [11] shows that this problem is NP-hard. So instead of solving VLC-OPT optimally, we will use DMMOPT to solve it approximately but in polynomial time.

D. Program Development

Following DMMOPT guidelines to construct a program that solves VLC-OPT, we first instantiate the Lagrangian (6) for this particular problem. To simplify the discussion, we will assume the search space of configurations $\mathcal{L}$ is the set that uses only table lookups and 1-bit programmed logic as decoding operators.[6]

To solve (6) for a particular value of λ, we first represent the set of VLCs in question as a binary tree, where nodes are numbered in post-order with root r. We define $f(i)$ as a function that returns the minimum Lagrangian cost, $H_{T(S)}(l)+\lambda R(l)$, for all configurations that correctly decode the set of VLCs rooted at node i.

To solve $f(i)$, we perform the following case analysis. At node i, we have two choices: i) perform a logic operation at node i with cost $w_i Q$; ii) perform a h-bit table lookup operation at node i with cost $w_i T(S) + \lambda 2^h$. The minimum of these two choices for all possible table height plus the recursive cost of the children nodes will be the cost of the function at node i:

$$f(i) = \min\left\{ w_i Q + \sum_{j\in L_{1,i}} f(j), \min_{1\le h\le H_i}\left[w_i T(S) + \lambda 2^h + \sum_{j\in L_{h,i}} f(j)\right]\right\} \tag{9}$$

where H_i is the height of binary tree rooted at node i, and $L_{h,i}$ is the set of nodes at height h of tree rooted at node i. We note that there are overlapping subproblems when solving $f(r)$ using (9). For example, if s is a children node of r and t is a children node of s, then $f(t)$ will be used in the calculation of $f(r)$ as well as the calculation of $f(s)$. To avoid solving the same subproblem more than once, we use a dynamic programming table $F[\,]$ of size $r*1$ to store the calculated values $f(i)$ for $i = 1 \ldots r$. Each time the function $f(i)$ is called, it first checks if the entry $F[i]$ has been filled. If it has, then $f(i)$ simply returns the value $F[i]$. Otherwise, it calculates the value using (9) and stores it in the table. After solving the Lagrangian problem using (9), we have a configuration, denoted by l^o, that minimizes the Lagrangian problem for a particular multiplier value λ.

1) Singular Value Search for VLC Decoding: As stated in step 2 of the iterative projection method, we need to intelligently search for multiplier values and solve (6) over and over again—this is *singular value search*, of which the general notion was discussed in Section III-C. To instantiate for VLC-OPT, we first observe from (9) that by construction, the optimal configuration has Lagrangian cost of form

$$f(i) = \sum_{x\in X} w_x T(S) + \sum_{x\in X} 2^{h_x}\lambda + \sum_{y\in Y} w_y Q \tag{10}$$

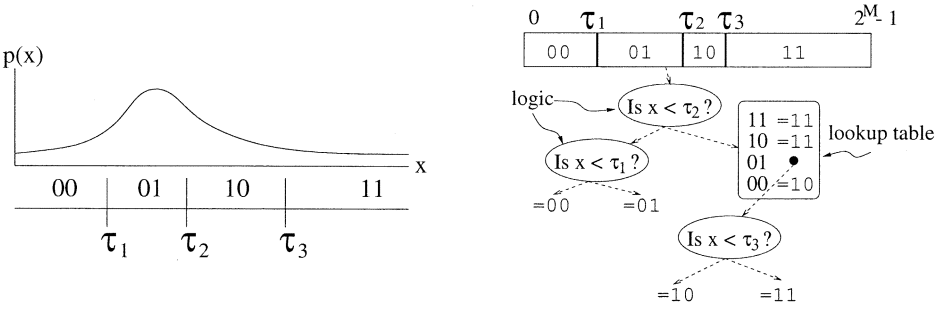
where X is the set of nodes performing table lookup operations, and Y is the set of nodes performing logic operations. Rewriting the equation yields a simpler representation—a linear function of λ with slope θ_i and y-intercept ζ_i:

$$f(i) = \zeta_i + \theta_i \lambda \tag{11}$$

$$\zeta_i = \sum_{x\in X} w_x T(S) + \sum_{y\in Y} w_y Q \tag{12}$$

$$\theta_i = \sum_{x\in X} 2^{h_x}. \tag{13}$$

[6]See [11] for a thorough discussion when the search space includes n-bit programmed logic and hash functions.



(a) (b)

Fig. 6. (a) Nonuniform SQ and (b) hybrid encoding implementation.

To find singular values, we first define $g(i)$ as a function that returns the next potential larger singular value, called *augmented singular value*, λ^+, for the tree rooted at node i. This value can be derived from one of two cases. First, it is the value at which a new configuration that uses a new operator at node i (for example, a logic operation at node i instead of a table lookup), in combination with the subconfigurations of the children nodes, becomes optimal as λ increases. Second, it is the value at which one of the descendant nodes of node i changes its optimal subconfiguration, affecting the optimality calculation for node i. $g(i)$ returns the smaller of these two values, as expressed in the following pseudo-code:

```
1. temp := I([ζ_i, θ_i], [w_i Q + Σ_{j∈L_{1,i}} ζ_j, Σ_{j∈L_{1,i}} θ_j])
if temp > λ, then g(i) := temp
//check config. w/ logic at node i
else g(i) := ∞
2. temp := min_{1≤h≤H_i}
  {I([ζ_i, θ_i], [w_i T(S) + Σ_{j∈L_{h,i}} ζ_j, 2^h + Σ_{j∈L_{h,i}} θ_j])}
if temp > λ & temp < g(i), then g(i) := temp
//check config. w/ lookup table at node i
3. temp := min_{j∈L_{1,i}} g(j)
if temp > λ & temp < g(i), then g(i) := temp
//check the potential s.v.'s of
 children nodes
```

where function $I([\zeta_1, \theta_1], [\zeta_2, \theta_2])$ takes in the slopes θ's and y-intercepts ζ's of two lines, and returns the intersection point. If they are parallel lines, it returns ∞.

$g(i)$ can be tabulated as (9) is being solved. The slope θ_i and y-intercept ζ_i of node i are calculated using (11) after the optimal configuration is found for tree rooted at i, and they are then stored in dynamic programming table $\theta[\,]$ and $\zeta[\,]$, similar to table $F[\,]$ used in solving (9). By calling $f(r)$ to solve (6) then $g(r)$ to find the augmented multiplier λ^+ repeatedly, we will terminate with the solution required in step 2 of the iterative projection method. If $R(l^o) = S^i$, $H(l^o)$ is a sample point on $H'(S)$. If not, we perform constraint shift (step 3) and repeat the procedure.

V. Scalar and Vector Quantizer

We now focus on the second instantiation of DMMOPT: scalar and vector quantizer encoding.

A. Scalar Quantizer Encoding

We begin with the nonuniform SQ encoding problem. We first define the search space of configurations $\mathcal{L}$, then define the optimization problem formally (SQE-OPT). Instead of discussing the development of the entire program using DMMOPT, as we did for VLC-OPT, we focus only on solving the Lagrangian (6) given λ. The other half of the puzzle, singular value search used in step 2 of the iterative projection method, can be found in [12].

1) Problem Formulation: Very often a representation of a signal in a computer needs to be compressed for space-limited storage or for bandwidth-limited transmission. One compression technique that can perform this many-to-few bits mapping of signal is the nonuniform M-to-N bit SQ, where a scalar quantity of M bits is mapped to one of 2^N partitions. Fig. 6(a) shows an example of a M-to-2 bit nonuniform SQ. The optimal design of nonuniform SQs—the selection of partition boundary set $\Gamma = \{\tau_1, \ldots, \tau_{2^N-1}\}$ that minimizes distortion of reconstruction signals—is well-studied [20]. The resulting N-bit quantizer is commonly called the *Lloyd-Max Quantizer*.

Finding an efficient implementation for the M-to-N-bit SQ encoder can involve a tradeoff between computation and memory. Two simple encoding implementations illustrate the extremes of computation and memory tradeoffs. The first one minimizes computations by performing a single M-bit table lookup, where the resulting table entry contains the corresponding N-bit partition index. This requires a memory store of size 2^M. An alternative implementation minimizes memory usage by asking a sequence of logic statements "*Is* $x < \tau_i$*?*" until the correct partition has been identified. This corresponds to a binary decision tree of height $h \geq N$. A natural question is: what is the optimal hybrid scheme, using a combination of lookup tables and logic, that minimizes the average encoding time? An example of a hybrid implementation is shown in Fig. 6(b).

Having described the search space $\mathcal{L}$, we formally define the optimization problem of finding the optimal hybrid implementation for the M-to-N bit SQ encoder, denoted as SQE-OPT, as follows:

Optimal SQ encode problem: SQE-OPT

Given search space $\mathcal{L}$, what is the fastest configuration $l \in \mathcal{L}$ of a M-to-N bit SQ encoder, given input x distribution $p(x)$, partition boundary set $\Gamma = \{\tau_1, \ldots, \tau_{2^N-1}\}$, and parameters of DMM?

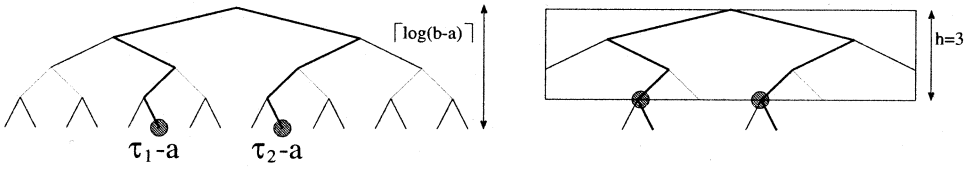


Fig. 7. Tree pruning example.

2) Program Development: We discuss instantiation of (6) for SQE-OPT in this section. We begin with the definition of the cost function that we are minimizing: let $f(a,b)$ be the minimum Lagrangian encoding cost (6)—$\min_{l\in\mathcal{L}} H_{T(S)}(l) + \lambda R(l)$—given input $x \in [a,b)$. The optimal initial operation for this input range can potentially be a logic or table lookup, resulting in cost $f_l(a,b)$ or $f_t(a,b)$, respectively:

$$f(a,b) = \min\{f_l(a,b),\ f_t(a,b)\} \tag{14}$$

For the logic case, we can choose among all τ_i values that are in range (a, b) to check against input x. The result of the check is a partition of the original interval into $[a,\tau_i)$ and $[\tau_i,b)$. Let $p(a,b)$ denote the probability that $x \in [a,b)$, and Q denote the cost of a logic operation. We can write $f_l(a,b)$ as

$$f_l(a,b) = p(a,b)Q + \min_{\tau_i\in(a,b)} [f(a,\tau_i) + f(\tau_i,b)]. \tag{15}$$

For the table lookup case, there is an initial lookup cost of $p(a,b)T(S)$. To minimize table size, $x-a$ is used as the index into the lookup table. Hence the index used in a h-bit table lookup operation is the h left-most bits of $x-a$. The largest value $x-a$ can take on, given $x \in [a,b)$, is $b-a-1$. Hence, the number of bits needed to describe $x-a$, or the maximum height of a lookup table, is $\lceil \log_2(b-a)\rceil$. For each table height h, the table operation divides the range $[a,b)$ into smaller ranges of width $m = 2^{\lceil\log_2(b-a)\rceil - h}$ each. The number of these smaller ranges, n, is determined by the largest number that the h most significant bits of $x-a$ can take on. The associated penalty $\lambda R(l)$ in (6) is therefore λn. The following equations formalize this (see (16), shown at the bottom of the page).The base case of the recursion is when there is no τ_i in range (a, b), meaning the input x can only be in one partition:

$$f(a,b) = 0 \qquad \text{if } \nexists \tau_i \in (a,b). \tag{17}$$

The value of $f(0,2^M)$ then yields l^o, the optimal solution to (6) given λ.

3) Tree Pruning: While we can solve (6) given λ with call to $f(0,2^M)$ using recursive calls (14)–(16), the running time is exponential—the call has 2^M recursive calls when a M-bit table lookup operation is tested for $f_t(0,2^M)$. This means a single rexecution of (16) has running time $O(2^M M)$. However, we can alter the program to reduce its complexity by pruning off some of the recursive calls in (16). When performing an h-bit table lookup for a given range $[a,b)$, there will be n recursive calls according to (16) corresponding to n branches of a binary tree—the leaves of the tree are the possible values of $x-a$. Fig. 7(a) shows a binary tree representation of numbers in range $[a,b)$—from most significant bit to least significant bit—where $\lceil\log_2(b-a)\rceil = 4$. It also highlights the branches that corresponds to each $\tau_i \in [a,b)$. We call them τ-*branches*.

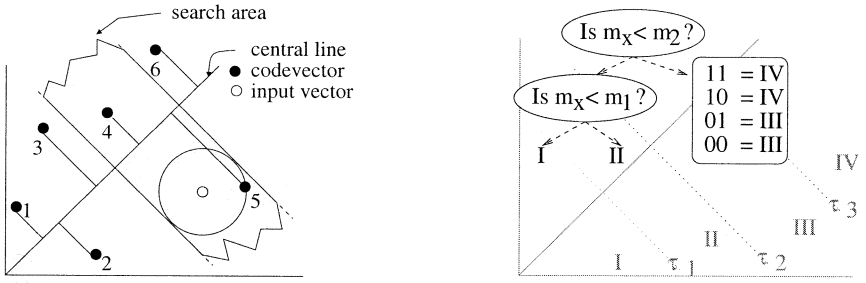
If we perform a 3-bit table lookup on the same range $[a,b)$, as shown in Fig. 7(b), we see that unless input x follows one of the two τ-branches, we know immediately which partition the input falls into. If input x follows one of the two τ-branches, then further operations are needed to determine the correct partitions. We can generalize the above observation and say that the only recursive calls needed in (16) are these τ-branches. So if we prune off the non-τ-branches during execution of (16), the complexity of (16) is $O(2^N M)$; it is now polynomial in size of the input.

B. Vector Quantizer

Instead of developing a new optimization, we leverage on the program we developed for SQE-OPT and use it to speed up a pre-processing step of an established VQ encoding technique, called *equal-average nearest neighbor search* (ENNS). We first describe how ENNS works, then discuss how the program developed for SQE-OPT can be used to improve ENNS.

1) Equal-Average Nearest Neighbor Search: ENNS [21] has been shown to lower unstructured VQ encoder's complexity in the average case for image data. The key observation is that there are strong correlations among input vector's individual components for image data. As a result, the majority of the input vectors are distributed along the central line $l = \{\mathbf{x} | x_1 = \ldots = x_k\}$. ENNS proposes that we first presort

$$\begin{aligned} f_t(a,b) = {} & p(a,b)T(S) + \min_{1\le h\le\lceil\log_2(b-a)\rceil} \\ & \left[\lambda n + \sum_{i=1}^{n-1} f\left(a+m(i-1), a+m(i)\right) + f\left(a+m(n-1), b\right)\right] \\ m = {} & 2^{\lceil\log_2(b-a)\rceil - h} \qquad n = \left\lfloor \frac{b-a-1}{m} \right\rfloor + 1. \end{aligned} \tag{16}$$



(a) (b)

Fig. 8. (a) Equal-average VQ encoding and (b) hybrid encoding implementation.

the codevectors according to their means, then during actual algorithm execution to find the nearest neighbor to input vector $\mathbf{x}$, we can successively eliminate potential nearest neighbors by using this bound:

$$d(\mathbf{x},\mathbf{y}) \geq \sqrt{k}|m_{\mathbf{x}} - m_{\mathbf{y}}| \tag{18}$$

where $d(\mathbf{x},\mathbf{y})$ is the l_2 distance between input vector $\mathbf{x}$ and potential nearest neighbor $\mathbf{y}$, k is the number of dimensions, and $m_{\mathbf{x}}$ and $m_{\mathbf{y}}$ are mean of $\mathbf{x}$ and $\mathbf{y}$ respectively.

An example is shown in Fig. 8(a), where input vector $\mathbf{x}$ is matched against codevectors $\{\mathbf{y}_1 \ldots \mathbf{y}_6\}$ in two-dimensional space. We first test codevector $\mathbf{y}_5$ and compute $d(\mathbf{x},\mathbf{y}_5)$. We can then eliminate any vector $\mathbf{y}_i$ whose mean $m_{\mathbf{y}_i}$ is such that $\sqrt{k}|m_{\mathbf{x}} - m_{\mathbf{y}_i}| \geq d(\mathbf{x},\mathbf{y}_5)$. Geometrically, we eliminate all codevectors that lie outside the strip that encloses the circle in Fig. 8(a). In this example, we eliminate $\mathbf{y}_1$, $\mathbf{y}_2$, $\mathbf{y}_3$ and $\mathbf{y}_6$.

For ENNS to be most effective, the initial candidate codevector should have mean closest to the input vector. To this end, ENNS uses a binary decision tree to first find this closest-input-mean codevector. To speed up this initial search, we use the program for SQE-OPT to find a near-optimal implementation that finds this closest-input-mean codevector.

In order to use the program for SQE-OPT to generate a near-optimal implementation that finds the closest-input-mean codevector, we need to supply the inputs of the program: parameters of DMM, partition boundary set, and input distribution. Parameters of DMM is the same as the SQ case. The partition boundary set Γ is found by partitioning the space of input mean into bins so that if the input mean falls into a bin, then the closest-input-mean codevector is the codevector associated with the bin. The input distribution is the probability distribution of the input mean.

VI. Results

A. IP Address Lookup

In this section, we demonstrate the efficacy of the generated configuration for IP address lookup using the program described in Section IV. We obtained a routing table with 2638 prefixes from the Palo Alto Internet Exchange (PAIX) from [19] on June 19, 1998. We were unable to obtain the statistics of these prefixes, so we modeled the workload with two prefix probability distributions for our simulation: 1) *Equal*, where all prefixes are equally probable; and 2) *Scaled*, where an h-bit prefix is twice as likely as an $h+1$-bit prefix (so longer prefixes are less likely than shorter ones). We additionally assumed the recurring probability p in the Markov model is 0.67, resulting in three consecutive packets on average with the same longest matched prefix before another prefix is randomly selected again. The measurements were taken on a Pentium II 266 MHz, with L1 cache 16 kBytes and L2 cache 512 kBytes. To match this environment, we estimate the machine model parameters to be $(T_1, T_2, Q) = (2, 4, 3)$, where the unit is number of processor clock cycles.

To find a near-optimal configuration, we first constructed the Lagrangian sampled function $H'(S)$ and found the minimum sample point empirically. We constructed two such functions, each has a search space $\mathcal{L}$ reflecting different collection of decoding operators available. For the first function, we used only one decoding operator—table lookup up to any height. This is the top curve in Fig. 9. Following the definition of $H'(S)$ in (4), the cost unit on the y-axis is time/clock cycles. For the second function, $\mathcal{L}$ includes an additional decoding operator—optimal sequential programmed logic of up to height 4. The first function

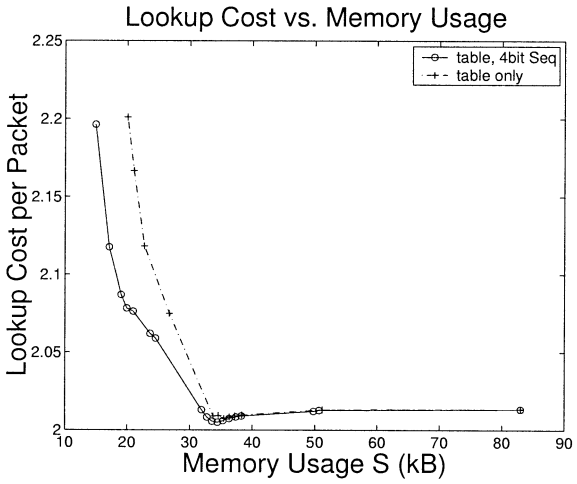


Fig. 9. Sampling functions $H'(S)$ for 2 different Search Spaces $\mathcal{L}$.

Srinivasan & Varghese [18]	3.902 mil. lookups/s
Cheung & McCanne 99 [10]	4.546 mil. lookups/s
Lookup Tables only	4.938 mil. lookups/s
Tables + 4bit Seq. Logic	5.000 mil. lookups/s

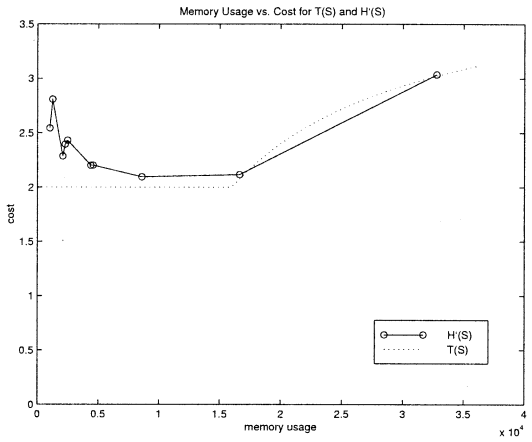
Fig. 10. Results for equal probability prefixes.

Srinivasan & Varghese [18]	3.961 mil. lookups/s
Cheung & McCanne 99 [10]	7.143 mil. lookups/s
Tables + 4bit Seq. Logic	7.273 mil. lookups/s

Fig. 11. Results for scaled probability prefixes.

Parameters	Values
S_1	$16k$
T_1	2 cycles
S_2	∞
T_2	4 cycles
Q	3 cycles
$p(x)$	Gaussian 1: $N(8000, 400^2)$
	Gaussian 2: $N(8000, 1600^2)$

(a)



(b)

Fig. 12. Parameters and sampling functions for SQ experiment. (a) DMM parameters and (b) T(S) and H'(S).

is above the second one for all S; this agrees with our intuition since $\mathcal{L}$ of the second function includes $\mathcal{L}$ of the first one.

To test the synthesized configurations, we compared the performance of our configurations against two other algorithms: 1) a lookup table design algorithm known as *controlled prefix expansion* presented in [18] which minimizes worst case instead of average case, and 2) a similar optimization procedure we developed in 1999 that used a simpler static memory model [10]. Using the PAIX prefix routing table and the two probability distributions as previously discussed, we simulated a workload of 10 million IP addresses using the Markov model discussed in Section IV-A. For each algorithm, we repeated the simulation 40 times to find the average decoding speed.

Fig. 10 shows that our synthesized configuration of the second function outperforms Srinivasan & Varghese by 28.2%, and Cheung & McCanne 99 by 10.0%. Note again that the configuration of the second function, "Tables + 4 bit Seq. Logic," is faster than the configuration of the first function.

We conducted a second set of measurements using a workload generated from the scaled probability distribution in Fig. 11. The optimal configuration of the second function outperforms Srinivasan & Varghese by 83.6%, and Cheung & McCanne 99 by 1.82%. We conjecture the reason for the dramatic improvement over Srinivasan & Varghese to be the following: because the statistics are very skewed—long prefixes are very unlikely, and so improving the lookup speed of shorter ones drastically improves the average decoding speed. As a result, the worst-case optimal solution, which is largely concerned with longer prefixes, is far from the average-case optimal solution.

B. Scalar Quantizer

To evaluate the performance of our program for SQ, we conducted experiments to compare our generated configuration to an implementation that use a binary decision tree to encode SQ [20]. For our experiments, we use parameters in Fig. 12(a), which are estimates of our test machine, a Pentium II 266 MHz processor. It has a 16-kbyte L1 cache (50-50 split of the 32-kbyte data-instruction cache). The two input distributions $p(x)$ are arbitrarily chosen Gaussian distributions.

p(x)	M	N	algorithm	speed
Gaussian 1	15	2	logic-only	3.947 mil/s
Gaussian 1	15	2	hybrid	4.651 mil/s
Gaussian 1	15	4	logic-only	2.469 mil/s
Gaussian 1	15	4	hybrid	4.545 mil/s
Gaussian 2	15	2	logic-only	3.738 mil/s
Gaussian 2	15	2	hybrid	4.790 mil/s
Gaussian 2	15	4	logic-only	2.484 mil/s
Gaussian 2	15	4	hybrid	4.597 mil/s

Fig. 13. Comparison of SQ encoders.

Using the chosen parameters, we generated $T(S)$ (bottom) and $H'(S)$ for 15-to-4 bit scalar quantizer, with input distribution Gaussian 2 (top) in Fig. 12(b). As in Fig. 9, the cost unit is time/clock cycle. The partition boundary set Γ was generated using Lloyd's algorithm.

To compare our hybrid table lookup-logic SQ encoder to a binary decision tree SQ encoder, we generated a workload of 20 million input samples according to the input distribution and encoded them 10 times with each algorithm to find an average speed for each case. See Fig. 13 for experimental results.

For the Gaussian 1 input distribution 15-to-2 bit (15-to-4 bit) SQ encoders, excluding I/O access time, we achieved a 17.84% speed improvement (84.08%) over the logic-only encoder. For the Gaussian 2 input distribution 15-to-2 bit (15-to-4 bit) SQ encoders, we see a 28.14% improvement (85.06%) over the logic-only encoder. As N increases, the improvement of hybrid encoders over logic-only encoders increases. This is expected, since the height of the binary decision tree for logic-only encoders is larger when as N increases.

C. Vector Quantizer

We performed experiments to show that ENNS has faster encoding speed when the configuration found by the program solving SQE-OPT is first used to find the codevector with the closest mean to input vector. To generate various VQ codebooks for testing, we used 512×512 gray scale images of `Lena`, `Baboon` and `Tiffany` as training data, and constructed codebooks of size 8, 16, 32, and 64 for dimension 4 using the generalized Lloyd algorithm [20]. We then compared the encoding

codebook size	algorithm	encoding time
8	logic-only	.280s/lena
8	hybrid	.255s/lena
16	logic-only	.465s/lena
16	hybrid	.440s/lena
32	logic-only	.465s/lena
32	hybrid	.440s/lena
64	logic-only	.935s/lena
64	hybrid	.895s/lena

Fig. 14. Comparison of VQ encoders.

speed of ENNS using a binary decision tree and ENNS using our generated configuration when encoding the Lena image. See Fig. 14 for experimental results. Excluding I/O access time, we achieved speed improvement of 9.83%, 5.67%, 6.65%, and 4.47% respectively for the four codebook sizes.

A few observations can be made. First, we observe that the improvement for VQ is not as drastic as SQ. This is expected, since we are speeding up only the initial search for closest codevector mean, and the VQ encoding algorithm needs to perform other tasks like computing distortion between input vector and potential candidate vectors. Second, we see that as the size of the codebook increases, the percentage improvement decreases. The reason is that ENNS is increasingly ineffective in ruling out candidate codevectors as the codebook size grows. The bulk of the computation then becomes the computations of distortion between input vector and candidate vectors, and the speed improvement of initial search for closest codevector mean is diminished in the overall picture.

VII. Conclusion and Future Work

In this paper, we presented a dynamic memory model and associated optimization framework that finds a near-optimal implementation within a search space of implementations by exploiting the computation-memory tradeoff of the underlying machine. We discussed two applications of the framework: fast IP address lookup, and fast nonuniform scalar quantizer and unconstrained vector quantizer encoding. In the results section, we have shown that there are noticeable improvements over competing techniques in both applications.

There are two different directions for possible future work. First, the memory model we constructed (DMM) is undeniably simple—it does not take into consideration of details of a particular processor's memory architecture, such as set associativity. Admittedly, the memory model was constructed with an eye to the basic input stochastic models of the two applications in the paper—a complex memory model is useless if the input model cannot take advantage of it. An interesting problem is how to create a more accurate memory model that corresponds well to a general class of input models, while keeping the resulting optimization problem tractable. Second, the applications of the framework we looked at lead naturally to obvious and easily defined search spaces of configurations $\mathcal{L}$'s, leading to tractable dynamic programming solutions. It would be interesting to look at other computation-intensive algorithms, such as motion estimation and packet classification, where defining the space $\mathcal{L}$ and finding the corresponding solution to the Lagrangian are much more challenging.

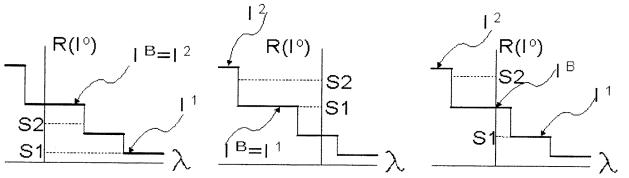


Fig. 15. Three cases for local error bound theorem: (a) case I, (b) case II, and (c) case III.

Appendix

We prove Theorem 3.1 in this section. We first show that we can always find l^B that satisfies $R(l^B) \geq S^2$ during singular value search for $S = S^1$—the first half of the first bounding condition.

1) Lemma 1: Given there are two Lagrangian sample points at S^1 and S^2 with corresponding solution l^1 and l^2, there exists an optimal solution l^B to (6) such that $R(l^B) \geq S^2$, during singular value search for $S = S^1$.

a) Proof 1: We prove the statement by contradiction: suppose l^T is the top step of $R(l^o(\lambda))$ and $R(l^T) < S^2$. That means there exists some multiplier value λ^o, such that $\forall \lambda \leq \lambda^o$:

$$H_{T(S^1)}(l^T) + \lambda R(l^T) \leq H_{T(S^1)}(l) + \lambda R(l) \quad \forall l \in \mathcal{L}. \tag{19}$$

By assumption, $R(l^T)$ is strictly smaller than $S^2 = R(l^2)$. So for some $\lambda^+ > 0$, $\lambda^+ R(l^T) < \lambda^+ R(l^2)$. Let $\lambda^- = -\lambda^+$. For λ^+ sufficiently large, we have

$$H_{T(S^1)}(l^T) + \lambda^- R(l^T) > H_{T(S^1)}(l^2) + \lambda^- R(l^2). \tag{20}$$

So $\nexists \lambda^o$ such that $\forall \lambda \leq \lambda^o$, l^T is the optimal solution, which is a contradiction. □

We are now ready to prove Theorem III.1. With the help of Lemma 1, we will prove the theorem by case analysis.

b) Proof: Let l^1 be the optimal solution corresponding to $H'(S^1)$. Let l^2 be the step on $R(l^o(\lambda))$ with $R(l^2)$ closest to S^2, such that $R(l^2) \geq S^2$—from Lemma 1, we know l^2 must exist. We claim that at least one of the following three cases must be true: 1) **Case I**: part of step at l^2, and step at l^1, occur at $\lambda \geq 0$, 2) **Case II**: step at l^2 and part of step at l^1 occur at $\lambda < 0$, 3) **Case III**: step at l^2 occurs at $\lambda \leq 0$ and step at l^1 occurs at $\lambda \geq 0$. From observing Fig. 15, this is true obviously. It is clear that for each case, there exists an optimal solution l^B to (6) such that one of the bounding conditions is satisfied: for Case I, we let $l^B = l^2$; for Case II, we let $l^B = l^1$; for Case III, we let l^B to be the step at $\lambda = 0$. We now prove that if one of the conditions is satisfied, then the error bound holds.

We first define two related optimization problems to (4) by constraint relaxations:

$$H'_{\leq}(S^*) = \min_{l \in \mathcal{L}} \{H_{T(S^*)}(l)\} \quad \text{s.t. } R(l) \leq S^* \tag{21}$$

$$H'_{\geq}(S^*) = \min_{l \in \mathcal{L}} \{H_{T(S^*)}(l)\} \quad \text{s.t. } R(l) \geq S^*. \tag{22}$$

Let $l^*_{\leq}$ and $l^*_{\geq}$ be the optimal solutions to (21) and (22), respectively. Since the search spaces for both problems are both supersets of (4), it is clear that $H'_{\leq}(S^*) \leq H'(S^*)$ and $H'_{\geq}(S^*) \leq H'(S^*)$. We now prove each of the three cases separately.

Case I: Given an optimal solution l^B to (6) for $\lambda \geq 0$, and $R(l^B) \geq S^2$. By optimality:

$$H_{T(S^1)}(l^B) + \lambda R(l^B) \leq H_{T(S^1)}(l) + \lambda R(l) \quad \forall l \in \mathcal{L} \tag{23}$$

$$\lambda \left[R(l^B) - R(l)\right] \leq H_{T(S^1)}(l) - H_{T(S^1)}(l^B). \tag{24}$$

If we let $l = l^*_{\leq}$, given $\lambda \geq 0$ and $R(l^*_{\leq}) \leq S^* \leq S^2$, term on left is nonnegative. Hence

$$H_{T(S^1)}(l^B) \leq H_{T(S^1)}\left(l^*_{\leq}\right) \leq H_{T(S^*)}\left(l^*_{\leq}\right) \tag{25}$$

$$\leq H_{T(S^*)}(l^*) = H'(S^*) \tag{26}$$

where the second inequality of (25) is true because $S^1 \leq S^*$ implies $T(S^1) \leq T(S^*)$ by nondecreasing property of $T(S)$, and, therefore $H_{T(S^1)}(l) \leq H_{T(S^*)}(l)\ \forall l \in \mathcal{L}$. Therefore, the error bound holds for Case I.

Case II: Given an optimal solution l^B to (6) for $\lambda < 0$, and $R(l^B) \leq S^1$. Following the same optimality argument, we again get (24). For $l = l^*_{\geq}$, we can again argue the left term is nonnegative, since the two products are strictly negative and nonpositive respectively. Hence

$$H_{T(S^1)}(l^B) \leq H_{T(S^1)}\left(l^*_{\geq}\right) \leq H_{T(S^*)}\left(l^*_{\geq}\right) \tag{27}$$

$$\leq H_{T(S^*)}(l^*) = H'(S^*). \tag{28}$$

Therefore, the error bound holds for Case II.

Case III: Given l^B is an optimal solution to (6) for $\lambda = 0$. Following the optimality argument, we again get (24). Now with $\lambda = 0$, we get

$$H_{T(S^1)}(l^B) \leq H_{T(S^1)}(l) \quad \forall l \in \mathcal{L}. \tag{29}$$

This is also true for the locally optimal solution l^*. Hence

$$H_{T(S^1)}(l^B) \leq H_{T(S^1)}(l^*) \leq H_{T(S^*)}(l^*) = H'(S^*). \tag{30}$$

Therefore, the bound holds for Case III. We have proven all cases, and so the theorem is proven. □

REFERENCES

[1] D. Engler and T. Proebsting, "DCG: an efficient, retargetable dynamic code generator," in *ASPLOS'94*, 1994.

[2] D. Engler, "VCODE: a retargetable, extensible, very fast dynamic code generation system," in *PLDI'96*, 1996.

[3] C. Gebotys, "A minimum-cost circulation approach to DSP address-code generation," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 726–741, June 1999.

[4] S. Bhattacharyya, R. Leupers, and P.Petter Marwedel, "Software synthesis and code generation for signal processing systems," *IEEE Trans. Circuits Syst. II*, vol. 47, pp. 849–875, Sept. 2000.

[5] K. Lengwehasatit and A. Ortega, "Distortion/decoding time tradeoffs in software DCT-based image coding," in *ICASSP'97*, 1997.

[6] V. Goyal and M. Vetterli, "Computation-distortion characteristics of block transform coding," in *ICIP'97*, 1997, pp. 2729–2732.

[7] D. Patterson and J. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 1997.

[8] C. Gebotys, "Low energy memory component design for cost-sensitive high performance embedded systems," in *Proc. Custom Integrated Circuits Conference*, 1996.

[9] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas, "L1 data cache decomposition for energy efficiency," in *ISLPED'2001*, 2001.

[10] G. Cheung and S. McCanne, "Optimal routing table design for IP address lookups under memory constraints," in *Infocom 99*, Mar. 1999.

[11] ——, "Dynamic memory model based framework for optimization of ip address lookup algorithms," in *ICNP'99*, Nov. 1999.

[12] ——, "Dynamic Memory Model Based Optimization of Scalar and Vector Quantizer Encoder,", Berkeley CS Tech. Rep. UCB/CSD-99-1085, Feb. 28, 2000.

[13] Y. Shoham and A. Gersho, "Efficient bit allocation for an arbitrary set of quantizers," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 36, pp. 1445–1453, Sept. 1988.

[14] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *SIGCOMM '97*, 1997, pp. 3–13.

[15] M. Waldvogel, G. Varghese, J. Turner, and B. Platter, "Scalable high speed ip routing lookups," in *SIGCOMM'97*, 1997.

[16] K.Keith Sklower, "A Tree-Based Routing Table for Berkeley UNIX," Tech. Rep., Univ. California, Berkeley.

[17] S. Nilsson and G. Karlsson, "Fast address lookup for internet routers," in *Int. Conf. Broadband Communication*, Apr. 1998.

[18] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," in *ACM Sigmetrics'98*.

[19] http://www.merit.edu/ipma [Online]

[20] A. Gersho and R. Gray, *Vector Quantization and Signal Compression*. Norwell, MA: Kluwer, 1992.

[21] L. Guan and M. Kamel, "Equal-average hyperplane partition method for vector quantization of image data," *Pattern Recognit. Lett.*, vol. 13, pp. 693–699, 1992.

[22] ftp://isdl.ee.washington.edu/pub/VQ/code/ [Online]

Gene Cheung (M'00) received the B.S. degree in electrical engineering from Cornell University, Ithaca, NY, in 1995, and the M.S. and Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1998 and 2000, respectively.

Since August 2000, he has been a Member of Technical Staff, Hewlett-Packard Laboratories Japan, Tokyo. His research interests include signal processing, computer networks and optimization.

Steven McCanne, photograph and biography not available at the time of publication.